Practical Verification Condition Generation for a Bytecode Language

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

January 21, 2015

Abstract

Automatic program verifiers typically generate verification conditions from the program and discharge them with an automated theorem prover. An important consideration is the manner in which program code and invariants are expressed. We have developed a bytecode language (similar, in spirit, to Java bytecode) on which verification is performed. This serves as both an intermediate language for use within the compiler, and a binary format with which dependencies (e.g. for libraries) can be resolved. Our bytecode language is a three-address code with semi-structured control-flow. Program code and invariants are represented uniformly to ensure bytecode programs are compact. In this paper, we present our bytecode language and outline a verification condition generator based on a path-sensitive forward-propagation algorithm.

1 Introduction

An important aspect of any tool for program verification is that of generating verification conditions [1]. This is the process by which programs expressed at the source-level are converted into one or more logical conditions whose validity determines the program's overall correctness. Verification condition generation must be performed on some abstraction of the program source. This could be, for example, a classical Abstract Syntax Tree (AST) representation, or on some kind of Intermediate Language (IL). Compilers face similar choices in determining when to perform the various stages of compilation (e.g. type checking, name resolution, code optimisation, etc). The trade-offs here are well known: operating on a high-level AST enables better feedback, but introduces more cases and implementation complexity. In contrast, operating on a low-level intermediate language simplifies a given stage's implementation, but makes it harder to provide useful feedback to the user (i.e. because of the greater separation between the original source and the representation being manipulated).

In a program verification tool, the verification condition generator is critical to the tool's overall soundness. This component is already complex in nature, and efforts to reduce this complexity can be enormously beneficial. As with compilers, one approach to tackling this problem is through an intermediate language. The authors of the widely acclaimed Spec# tool took this approach, and described it thusly [2]:

"*The gap between the program and the formula is bridged by translating the Spec# program into a much simpler program, much as a compiler bridges the gap between source program and machine code by translating into an intermediate representation*"

Indeed, the authors went on to state that introducing an intermediate language (i.e. Boogie) was "*the best and most far-reaching single design decision we made in implementing the Spec# verifier*" [2]. Another important benefit from intermediate languages are that different source languages may target them. This has been particularly evident for JVM bytecode, which now hosts a variety of additional source languages (e.g. Scala, Groovy, Kotlin, JRuby, etc). Such source languages benefit from the existing efforts to make the JVM a versatile execution platform. In a similar way, Boogie is also targeted by languages other than Spec# (e.g. Dafny [3, 4], VCC [5], etc). In this case, however, the benefits are from Boogie as a versatile verification platform.

Numerous pioneering verification tools have been developed over the last few decades, such as Gypsy [6], the Stanford Pascal Verifier [7], the Extended Static Checker for Modula-3 [8] and, more recently, ESC/Java [9], Spec# language [10, 11] and Dafny [3, 4] (to name but a few). Following in this line of work, we are developing a verifying compiler for the Whiley programming language [12]. Whiley is an imperative language designed to simplify verification. For example, Whiley uses unbound integer and rational arithmetic in place of e.g. IEEE 754 floating point, which is notoriously difficult to reason about [13]. Likewise, pure (i.e. mathematical) functions are distinguished from those which may have side-effects.

The Whiley compiler generates verification conditions from programs expressed in a bytecode language, rather than at the source-level. This offers a number of important benefits. Operating on a small bytecode language allows for a simpler verification condition generator, compared with operating at the source-level (which is much more expressive). Some constructs (e.g. **`break`** and **`continue`** statements) are particularly difficult to handle in a clean manner at the source level [14]. In the bytecode language, such constructs do not exist and are implemented uniformly using semi-structured control-flow. Finally, operating on a generic bytecode language offers flexibility in the future. For example, other programming languages may compile down to this bytecode language and can leverage our verification pipeline (i.e. as happened with Boogie). Likewise, future extensions to the Whiley source language may not always require changes to the verification pipeline (i.e. if they can be encoded using the existing bytecodes).

The contributions of this paper are:

1. We present a bytecode language into which program code and invariants can be uniformly compiled, and from which verification conditions can be safely extracted.

2. We outline a forward-propagation algorithm for generating verification conditions from programs written in our bytecode language.

2 Overview

Our goal with Whiley is to develop a language whose programs are both verifiable and can be compiled to run efficiently on different platforms, including a virtual machine. To that end, we have developed a bytecode language for representing Whiley programs in binary form. The bytecode language retains all type information and program invariants (particularly, pre- and post-conditions) necessary for compiling against. Thus, libraries and applications can be distributed in binary form (as with e.g. Java bytecode, MSIL and LLVM bitcode). The bytecode language also serves as a suitable intermediate language for different compiler back-ends and could be used for efficient interpretation and Just-In-Time compilation. In developing our bytecode language, we opted for a three-address code representation with semi-structured control-flow. This differs from e.g. Java bytecode (which adopts a stack-based representation) but is quite comparable with LLVM bitcode and the Dalvik Bytecode used for Android. A key requirement was to minimise the number of opcodes by representing program code and invariants uniformly within the same bytecode instruction set. This means that all program invariants are encoded using standard (imperative) bytecode instructions. To verify a program, the verification condition generator must reconstruct these invariants as logical conditions from their (imperative) bytecode encoding. Care must be taken in doing this to ensure the transformation is correct. In particular, the encoding of quantified formulas into imperative `forall` loops presents some challenges.

2.1 Basics

The *Whiley Intermediate Language (WyIL)* is a register-based intermediate language which loosely resembles Java Bytecode. The following illustrates a simple function in Whiley (left) which adds one to its parameter, alongside its WyIL representation (right):

```
function f(int x) => int:
    return x + 1
```

```
function f(int)=>int:
body:
    const %1 = 1
    add %2 = %0, %1
    return %2
```

A bytecode block (such as `body` above) has an unlimited register set available to it, and these are prefixed with `%` above (e.g. `%1`). As for JVM bytecode, the set of registers used in any given block is statically known and, furthermore, registers hold parameter values on entry (i.e. `%0` holds parameter `x` on entry). In the above example, the `const` bytecode loads an integer constant into register `%1`. The `add` bytecode performs addition on its two operands, assigning a result to a given register. The operand types for this bytecode must match and be either of type **`int`** or **`real`**. Finally, the **`return`** bytecode returns its (optional) operand.

Bytecode blocks, such as above, are statically typed and, as with the JVM, registers may have different types at different points. The latter increases flexibility and ties more closely the flow-typing discipline used in Whiley [15, 16]. Nevertheless, the type of each register at any given point in a block can be statically determined using a dataflow analysis similar to that found in the JVM (although there are some challenges here which we discuss elsewhere [17]).

2.2 Conditionals

Conditional constructs at the source-level are implemented using conditional and unconditional branches. These provide a form of *semi-structured* control-flow as they are restricted to forward branching only. That is, neither conditional nor unconditional branching instructions can form loops. The following illustrates:

```
function abs(int x) => int:
  if x >= 0:
    return x
  else:
    return -x
```

```
function abs(int) => int:
body:
  const %1 = 0
  ifge %0, %1 goto lab_0
  neg %0 = %0
.lab_0
  return %0
```

Here, the `ifge` bytecode branches to a label if its first operand is greater-or-equal to its second operand, whilst `neg` negates its operand and assigns this to a given register.

2.3 Loops

Since branching instructions cannot generate loops, special looping bytecodes are provided. This simplifies the process of establishing loop invariants and, at the same time, allows a variety of source-level statements to be encoded. The following illustrates:

```
function abs(int n) => int:
    int i = 0
    while i < n:
        i = i + 1
    return i
```

```
function abs(int) => int:
body:
  const %1 = 0
  loop modifies %1
    ifge %1, %0 goto lab_0
    const %2 = 1
    add %1 = %1, %2
.lab_0
  return %1
```

Here, the `loop` bytecode denotes a loop block. By itself, this says nothing about how the loop will be exited. In this example, a conditional branch is placed at the beginning of the loop to simulate the **`while`** loop. Equally, the condition could be placed at the end of the loop body to implement a **`do`**-**`while`** loop. The following illustrates:

```
function f(int n) => int:
    int i = 0
    do:
        i = i + 1
    while i < n
    return i
```

```
function f(int) => int:
code:
  const %1 = 0
  loop modifies %1
    const %4 = 1
    add %1 = %1, %4
    ifge %1, %0 goto lab_0
.lab_0
    return %1
```

This examples illustrates the flexibility of the `loop` bytecode — namely, that it can be used to implement multiple source-level constructs. This in turn simplifies the verifier by reducing the number of cases to be considered.

Finally, the loop bytecode always includes a *modifies* clause which indicates those registers which may be modified by the loop (in this case, register `%1` maybe modified). It is an error for any variable declared outside of the loop body to be modified within the loop without being present in this clause. The purpose of the modifies clause is to simplify the verification of variables *not* modified in the loop — specifically, so they can be omitted from the loop invariant (see §3.3). This follows Beckert *et al.* [18].

Forall Loops. The Whiley programming language supports first-class sets, maps and lists. In order to access the elements of a set, map or list, a special `forall` bytecode is provided. The following illustrates:

```
function sum({int} xs) => int:
    int r = 0
    for x in xs:
        r = r + x
    return r
```

```
function sum({int}) => int:
body:
  const %1 = 0
  forall %2 in %0 modifies %1
    add %1 = %1, %2
  return %1
```

Here, the type `{int}` indicates a set of integers. In the example, the `forall` bytecode declares register `%2` to iterate over the contents of register `%0` (i.e. the parameter `xs`) and, as before, register `%1` is declared in the modifies clause. As we'll see in §3.2, `forall` loops are also useful for implementing quantifiers in source-level assertions.

Break / Continue. The **`break`** and **`continue`** statements are also implemented using conditional and unconditional branches. As before, this simplifies verification condition generation as no additional support is required. The following illustrates:

```
function f([int] xs,int x)=>int:
  int i = 0
  while i < |xs|:
    if i >= x:
       break
    i = i + 1
  return i
```

```
function f([int],int)=>int:
body:
  const %2 = 0
  loop modifies %2
    lengthof %6 = %0
    ifge %2, %6 goto label0
    ifge %2, %1 goto label0
    const %10 = 1
    add %2 = %2, %10
.label0
  return %2
```

Here, the **break** statement is implemented using a single conditional branch. We can see that, from the verifier's perspective, the loop condition and the **break** can be handled uniformly. Finally, **continue** statements are implemented in much the same way, but branch to the end of the `loop` body.

3 Assertions

Verification of Whiley source files requires appropriate *assertions* (i.e. pre-/post-conditions, loop invariants, etc) are provided by the programmer. This limits verification to an intraprocedural activity which considerably improves tractability (and follows other tools, such as ESC/Java, Spec#, Dafny, etc). The assertions are themselves compiled into the bytecode language and (for the most part) reuse the existing bytecodes provided for implementing source-level statements and expressions. The bytecode language enables a wide range of source-level assertions to be encoded, making it extremely flexible.

3.1 Basics

Simple assertions are encoded directly using conditional and unconditional branching statements. A special `fail` bytecode is included to represent an assertion failure:

```
function f(int x) => (int r)
requires x >= 0
ensures r > 0:
  //
  return x + 1
```

```
function f(int) => int:
requires:
    const %1 = 0
    ifge %0, %1 goto label0
    fail
.label0
    return
    ...
```

Here, the **requires** and **ensures** clauses respectively represent the pre- and post-conditions of the function. The translation of the **requires** clause is shown on the right. We can see that the assertion is translated into a conditional branch over a `fail` bytecode. The intuition behind this is that the verifier will explore both execution paths and, for the failing path, must establish that it is unreachable (else report an error).

3.2 Quantifiers

Encoding of quantifiers at the source-level is achieved through the `forall` bytecode, which provides sufficient expressivity for a range of different quantifiers. For example:

```
function sum([int] ixs) => int
requires all {x in xs | x>=0}:
    ...
```

```
function sum([int]) => int:
requires:
  forall %1 in %0
    const %2 = 0
    ifge %1, %2 goto lab_1
    fail
  .lab_1
    nop // dummy
  return
  ...
```

Here, the universal quantifier **all** is used at the source level to require all elements of `xs` be greater-or-equal to zero. This is translated using the `forall` bytecode and makes use of a conditional branch and the `fail` bytecode. The dummy `nop` is included just to clarify the nesting level of `lab_1` — i.e. that it is within the body of the loop.

In addition to the **all** quantifier, the other supported quantifiers are: `some` (existential) and **no** (universal negated). These are encoded in a similar fashion, and the following illustrates the existential quantifier:

```
function sum([int] ixs) => int
requires some {x in xs | x>=0}:
    ...
```

```
function sum([int]) => int:
requires:
  forall %1 in %0
    const %2 = 0
    ifge %1, %2 goto lab_1
  fail
.lab_1
  return
  ...
```

In this case, the precondition for `sum` requires one or more elements of `xs` to be greater-or-equal to zero. This is implemented in a similar manner as before, except the loop terminates as soon a valid element is found. Furthermore, in the case the loop executes to completion (i.e. no matching elements were found), then the assertion fails.

Other useful source-level quantifiers could be supported through this scheme. For example, following Alloy [19], we could easily support quantifiers such as `one` (exactly one), `lone` (one or zero), etc.

3.3 Loop Invariants

Loop invariants are an important and challenging aspect of any verification system, and different systems have used a range of approaches for handling them. In particular, although Hoare logic provides a foundation for such tools, it leaves open many questions related to verification of loop invariants in real world systems. For example, how side effects should be handled, how break / continue statements should be handled, how do/while statements should be implemented, etc. Our bytecode language provides a flexible approach to dealing with such issues that supports a wide range of common approaches.

3.3.1 While Loops.

While loops illustrate the basic ideas behind the encoding of loop invariants in our bytecode language, and serve as a starting point for the remainder. The following illustrates a simple loop:

```
function f(int n) => (int r)
requires n >= 0
ensures r >= 0:
  //
  int i = 0
  while i < n where i >= 0:
      i = i + n
  //
  return i
```

```
function f(int) => int:
  ...
body:
  const %1 = 0
  loop modifies %1
    invariant:
      const %2 = 0
      ifge %1, %2 goto lab_0
      fail
    .lab_0
      return
    ifge %1, %0 goto lab_1
    add %1 = %1, %0
  .lab_1
  return %1
```

Here, we can see a special `invariant` block is used to represent the loop invariant. This block may be positioned anywhere within the loop itself, which enables a wide range of looping constructs to be encoded. In this case, the `invariant` block is placed at the start of the loop body, before even the condition. This reflects the usual interpretation of loop invariants, where the condition may rely on the loop invariant to hold but not vice-versa [20].

The loop invariant must hold on the first iteration of the loop (base case) and, assuming it held on the last iteration, must be shown to hold on this iteration (inductive case). The exact position of the invariant affects how this is interpreted (more later). This flexibility of position allows different source-level looping constructs to be represented, as well as more exotic approaches to loop invariants. For example, in Frama-C (an extension of C with support for assertions) multiple loop invariants may be specified at different positions within the loop [21, 22].

3.3.2 Do/While Loops.

A useful example which illustrates the value of the `invariant` block is the do-while loop:

```
method f(int x) => int
requires x >= 2:
  //
  int i = 0
  do:
      i = i + 1
  while (i+1) < x where i > 0
  ...
```

```
function f(int) => int:
  ...
body:
  const %1 = 0
  loop modifies %1
    add %1 = %1, %0
    invariant:
      const %2 = 0
      ifge %1, %2 goto lab_0
      fail
    .lab_0
      return
    ifge %1, %0 goto lab_1
.lab_1
  ...
```

This example is interesting because it highlights the difference between the while and do-while loops. Specifically, on entry to a while loop the loop invariant must hold. However, on entry to a do-while loop this is not the case — rather, the loop invariant must hold at the end of the first iteration. To implement this the `invariant` block is placed at the end of the loop body, but before the loop condition. This allows the condition to rely on the loop invariant, which is consistent with the treatment of while loops. Observe that, if one were compiling a language where the semantics for do-while loops were such that the loop invariant must hold on entry to the loop, then this could easily be accommodated by placing the `invariant` at a different position.

3.3.3 Break and Continue

The treatment of **break** and **continue** statement varies between different tools. In particular, an important question is whether or not the loop invariant must hold at the point of a **break**. This is often seen as necessary to obtain the usual guarantees of Hoare logic — namely, that the loop invariant holds after the loop has finished. Whiley, like a number of tools (including Dafny and Frama-C) has adopted an alternative approach. Specifically, that the disjunction of the loop invariant and the strongest assertion at the point of the break hold after the loop. The following illustrates:

```
method f(int n) => (int r)
requires n >= 0
ensures r == n || r == 10:
  //
  int i = 0
  while i < n where i <= n:
    if i == 10:
        break
    i = i + 1
  //
  return i
```

```
function f(int) => int:
requires:
  ...
body:
  const %1 = 0
  loop modifies %1
    invariant:
      ifle %1, %0 goto lab_0
      fail
    .lab_0
      return
    ifge %1, %0 goto lab_1
    const %2 = 10
    ifeq %1, %2 goto lab_1
    add %1 = %1, %0
.lab_1
  return %1
```

Intuitively, the verifier takes what is known at the point of a branch exiting the loop and carries this forward through the remainder of the function. Thus, what is known after the loop is simply a product of where the `invariant` is positioned in relation to those branches which exit the loop.

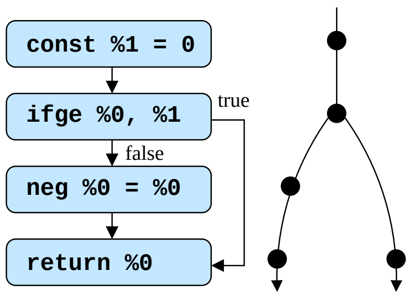
4 Verification Condition Generation

We now outline the algorithm for generating verification conditions from our bytecode representation. In doing this, we assume an appropriate target language for verification conditions which is some variant of first-order logic. In practice, this could be the input language for an SMT solver, such as SMT-LIB [23]. The algorithm operates in a path-sensitive forward propagation style. Although this can potentially result in large number of paths being explored, we have not encountered significant problems in practice. Furthermore, this approach means the verifier could report path-sensitive error messages which might be considerably more instructive for users (although we have not investigated this yet).

4.1 Overview

The verification condition generator traverses the control-flow graph of the bytecode function in a path-sensitive fashion. When a branching bytecode is encountered, the generator forks and proceeds independently down each path. At certain points during this traversal, verification conditions are generated from the current state and checked independently by the theorem prover. For example, upon reaching a **return** bytecode a verification condition is generated to ensure the function's post-condition holds. The following illustrates:

```
function abs(int) => int:
ensures:
    ...
body:
    const %1 = 0
    ifge %0, %1 goto lab_1
    neg %0 = %0
.lab_1
    return %0
```



Here, the generator would explore two distinct paths through the function and, for each, generate separate verification conditions at the **return** bytecode (i.e. to enforce the post-condition). In doing this, the generator accumulates knowledge about the current path taken through conditional branches and the effect bytecodes have on registers. Thus, reaching the **return** bytecode through the true branch of the conditional might yield the state $\%1_0 = 0 \wedge \%0_0 \geq \%1_0$ (ignore the subscripts for now). In contrast, traversing the false branch might yield $\%1_0 = 0 \wedge \%0_0 < \%1_0 \wedge \%0_1 = -\%0_0$.

Subscripts are used in intermediate generator states to denote the current assignment of variables (which is similar, in some ways, to static single assignment form [24]). In any given state, the greatest subscript for a given register denotes the "current" value, whilst lower subscripts denote "earlier" values. For example, $\%0_1$ represents the value of register $\%0$ after its assignment by the `neg` bytecode, whilst $\%0_0$ represents its value before that point. Observe that we cannot discard earlier values since they may contain important and relevant information (such as on the false branch above).

4.2 Preconditions and Postconditions

Postconditions generate *assertions* (i.e. verification conditions), whilst preconditions generate *assumptions* (i.e. initial states). A function's postcondition is traversed when a **return** bytecode is encountered to generate appropriate verification conditions. To do this, the generator first instantiates the postcondition by substituting the return value and any parameters used in the post-condition appropriately. Then, the generator traverses the post-condition using the current state. When a `fail` bytecode is encountered, it emits a verification condition which consists of exactly the current state. This establishes that the current state is unreachable or, if not, that an error exists.

The generator forms its initial state for the function by traversing the precondition. This is done slightly differently from the postcondition, since it is an assumption. Specifically, when a `fail` bytecode is encountered, the current state is simply dropped (i.e. rather than emitting a verification condition). Thus, those states which survive and exit the precondition form the assumptions fed into the start of the function body proper.

4.3 Loop Invariants

The treatment of loops and loop invariants is, as expected, slightly more involved. We begin by outlining a rough approach and then consider how to generalise it. We'll assume for now that the `invariant` block is located at the beginning of the loop block, so that no bytecodes in the loop precede it. For example:

```
...
while i < n where i >= 0:
    i = i + n
...
```

```
...
loop modifies %1
  invariant:
    ...
  ifge %1, %0 goto lab_1
  ...
.lab_1
...
```

In this case, the treatment of the invariant is relatively straightforward. When the generator reaches the start of the loop block, it traverses the loop invariant in a similar way as for a postcondition and any verification conditions emitted then enforce the invariant on entry. At this point, the generator prepares to traverse the loop by sending any variables modified in the loop to *havoc* (this is the reason for having an explicit `modifies` clause). This is achieved by incrementing the suffix for each such variable, which disconnects the current knowledge of the variable from that prior to the loop. This is necessary as the only knowledge we can retain about loop modified variables must be encoded in the invariant [18]. The generator then traverses the loop invariant in a similar way as for traversing a precondition to effectively assume the invariant. At this point, the generator traverses the loop body in the usual manner and, when the end is reached, will traverse the invariant again to assert it. In between the start and end of the loop multiple branches may be encountered which exit the loop. These are treated in the normal fashion with the generator forking to follow those branches without taking any other special action. In the case of our loop above, this means the assumed loop invariant (with the branch condition) carry forward directly out of the loop as expected. In this way, any number of branches which exit the loop are handled in a uniform fashion.

We now return to consider the general case where the loop invariant may be located at any point within the loop body. The treatment is essentially the same, though is somewhat less intuitive. Specifically, the generator operates in two phases as before. Initially, it proceeds upto the loop invariant and asserts it based on the state carried forward from before the loop. Then, it assumes the invariant and proceeds through and around the body until the loop invariant is reach again. In the specific case where the invariant occurs first in the loop, this has exactly the same effect as described above. Another way to think of this, is that the loop body is partially unrolled so that the invariant is now located at the start of the remaining loop.

4.4 Quantifiers

Quantifiers also present a challenge since these are encoded as `forall` loops and, hence, quantified expressions must be extracted from them. Furthermore, since a single kind of bytecode is used to encode all source-level quantifiers, it must be possible to extract a range of quantifiers as well. This requires careful tracking of which parts of the current state are from within the `forall` body, and those which came from before. The intuition is that those parts of the state generated within the `forall` body will form the body of the quantified expression. Then, the manner in which the current path exits the `forall` body determines which kind of quantifier is used. The following two examples illustrates the different ways quantifiers are generated:

```
requires:
  forall %1 in %0:
      const %2 = 0
      ifge %1, %2 lab_1
      fail
  .lab_1
      nop // dummy
  return
```

```
requires:
  forall %1 in %0:
      const %2 = 0
      ifge %1, %2 lab_1
  fail
.lab_1
  return
```

The left example represents a precondition such as “`all {x in xs | x >= 0}`”, whilst the right one such as “`some {x in xs | x >= 0}`”. The intuition is that there are two kinds of path which exit a `forall` bytecode. The first represents the “normal” termination of the loop (i.e. where all elements are iterated), whilst the second represents “abnormal” termination (i.e. where a branch exits the loop before iteration is completed). In the former case, a universal quantifier is constructed for the state generated within the body and, in the latter case, an existential quantifier is generated.

A key challenge faced in extracting quantifiers from `forall` loops is the presence of registers local to a loop. To understand why, consider again the left example above. Based on our description thus far, one might expect the state generated at the end of the loop to be: $\%2_0 = 0 \land \%1_0 >= \%2_0$.

But, how to universally quantify this? For example, $\forall \%1_0 \in \%0_0, \%2_0.(\%2_0 = 0 \wedge \%1_0 >= \%2_0)$ is clearly incorrect as $\forall \%2_0.(\%2_0 = 0)$ does not hold in general. The problem here is that register $\%2$ is *local* to the loop and, hence, should not be universally quantified at all. One option is to existentially quantifying all such local variables. However, reducing the number of quantifiers generated from expressions is important to improve success rates with the automated theorem prover[1]. Our solution is simpler, and doesn't require additional quantifiers. Specifically, whenever a register is defined we record the assigned expression and this is then substituted when a subsequent use of that variable is encountered. This ensures all generated states are purely in terms of the function's arguments and/or other declared variables (e.g. the index variable of a `forall` bytecode). Furthermore, since all states explored by the generator represent unique paths through the control-flow graph, situations where a variable has multiple definitions cannot arise.

Finally, we note that the special treatment of `forall` loops described above is used selectively within assertion blocks only (i.e. **`requires`**, **`ensures`**, `invariant`, etc). This prevents quantified expressions being generated unexpectedly from general loops, in particular those which e.g. may mutate non-local registers, etc.

5 Related Work

Perhaps the most comparable work is that of the Boogie Intermediate Language (Boogie IL) [25], which was developed as part of the Spec# project [2]. The goal of Boogie was to provide an intermediate target for verification which more closely resembles a programming language (in fact, Boogie shares some similarity with Dijkstra's language of guarded commands [26]). This contrasts with the common approach of generating verification conditions directly in the language of the theorem prover (e.g. SMT-LIB [23]), which usually resembles first-order logic. Such languages are much more "low-level" than Boogie and, hence, leave more work for the language implementer. Specifically, Boogie handles the generation of verification conditions over a function's control-flow graph. In contrast, when using SMT-LIB directly, one must implement this critical component as well. Boogie was considered a resounding success, and has been subsequently used as the verification "back-end" for other languages and systems, including Dafny [4], VCC [5], and more (e.g. [27, 28]).

In a similar fashion, the ESC/Java tool first translated functions into a guarded-command language, before generating verification conditions [9]. Guarded commands are another useful and interesting intermediate language for a verifying compiler [29]. Such a language typically consists of assignment, `assume` and **`assert`** statements as well as non-deterministic choice (amongst other things). This is much simpler than the original source language, but may encode a wide variety of source-level constructs. In particular, a function's potentially cyclic control-flow graph is translated into an acyclic guarded command program [30]. Loops are handled by removing back edges and assuming the invariant at the beginning of the body, and asserting it at the end. Thus, whilst guarded commands provide a useful intermediate step for verification, they still remain much "lower level" than the bytecode language presented here.

Barnett and Leino extended the basic use of guarded commands to support a so-called "passive form" where assignments are converted into `assume` statements [30]. The purpose of this is to reduce the size of generated verification conditions. Furthermore, their approach supports unstructured control-flow by introducing auxiliary variables to flag when a given set of instructions is active. Furthermore, loops identified from back-edges in the control-flow graph.

Another relevant work is that of Chalin, who examines the verification of real-world loops which contain unstructured control-flow (i.e. break/continue statements) and conditions with potential side-effects [20]. For example, a loop condition which contains a post-increment operator matches this form. Chalin identifies that "*40% of the 1500 loops in the Eclipse JDT (an industrial grade open source Java compiler) have conditions with side-effects, and/or bodies containing breaks, continue or return statements*". His approach to verifying such loops is to adopt a variation on the classical loop rule of Hoare Logic. The approach proposed by Chalin is to move the position within the

[1]Note that, whilst existential quantifiers are not in general a problem for automated theorem provers, they easily become universal quantifiers when assertions are negated, and this can have significant negative consequences.

loop where the invariant must hold to just after the condition, rather than before it. Furthermore, in contrast to the approach taken in Whiley, Chalin argues that loop invariants should "*be asserted to hold no matter how a loop is exited (be it via a break or return)*". Thus, we can see that the approach of Chalin could easily be supported within our bytecode language, as the position of loop invariants is not fixed.

Bannwart and Müller present a Hoare-style logic for a sequential bytecode language similar to e.g. JVM Bytecode or MSIL [31]. The primary motivation for their work is to enable the verification of proof-carrying code. The unstructured nature of bytecode languages presented a key challenge. To handle this, instructions are treated individually and the correctness of methods defined in terms of all execution paths contained, with a fixed-point iteration being used to converge loops.

Finally, Quigley developed a program logic for Java Bytecode encoded using Isabelle [32]. The goal was to enable more aggressive compiler optimisations within the JVM. Again, a significant challenge was the lack of structured information about loops. Her approach was to identify common patterns which, unfortunately, does not generalise to arbitrary loops expressible in bytecode.

6 Conclusion

In this paper, we have presented a bytecode language for encoding statements, expressions and assertions and, furthermore, outlined an algorithm for generating verification conditions over it. The bytecode language has two key features. Firstly, it minimises the number of bytecode types necessary by encoding assertions within those imperative bytecodes already required for statements and expressions, and with only a single additional bytecode being added (i.e. `fail`). Secondly, the bytecode language presents a flexible compilation target for verifying compilers which supports a wide range of source-level constructs. A particularly unusual feature of the language is the ability to place the loop invariant anywhere within a loop, thus making it possible to accurately encode a range of source-level loop statements (e.g. while versus do/while, etc). In the future, we would like to formalise our verification condition generator and provide an accompanying proof of soundness (which was outside the scope of this paper).

References

[1] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011.

[2] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

[3] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 6217 of *LNCS*, pages 112–126. Springer-Verlag, 2010.

[4] K. Rustan M. Leino. Developing verified programs with Dafny. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 82–82. Springer-Verlag, 2012.

[5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics (TPHOL)*, pages 23–42, 2009.

[6] D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

[7] D. Luckham, SM German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, and W. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[8] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[10] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. Technical report, Microsoft Research, 2004.

[11] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[12] The Whiley Programming Language, http://whiley.org.

[13] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 358–372, 2007.

[14] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, pages 284–303, 2000.

[15] D. J. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Computer Science*, 279(1):47–59, 2011.

[16] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 335–354, 2013.

[17] D. J. Pearce. A calculus for constraint-based flow typing. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*, page Article 7, 2013.

[18] Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt. An improved rule for while loops in deductive program verification. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 315–329, 2005.

[19] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[20] Patrice Chalin. Adjusted verification rules for loops are more complete and give better diagnostics for less. In *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*, pages 317–324. IEEE Computer Society Press, 2009.

[21] ACSL: ANSI/ISO C specification language (version 1.8).

[22] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the Conference on Software Engineering and Formal Methods (SEFM)*, volume 7504 of *LNCS*, pages 233–247. Springer-Verlag, 2012.

[23] The SMT-LIB standard: Version 2.0.

[24] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 25–35, 1989.

[25] M. Barnett, B. Evan Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2006.

[26] Edsger W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[27] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPU-Verify: a verifier for GPU kernels. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 113–132. ACM Press, 2012.

[28] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *Foundations of Security Analysis and Design*, volume 5705 of *LNCS*, pages 195–222. Springer-Verlag, 2009.

[29] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (FTFJP)*, 1999.

[30] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 82–87. ACM Press, 2005.

[31] Fabian Bannwart and Peter Mller. A program logic for bytecode. *Electronic Notes in Computer Science*, 141(1):255 – 273, 2005.

[32] Claire L. Quigley. A programming logic for Java Bytecode programs. In *Proc. TPHOLs*, pages 41–54, 2003.